APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: **HEURISTIC FOR GENERATING OPPOSITE INFORMATION FOR INCLUSION IN FOCUS EVENTS**

APPLICANT: **Hania Gajewska and David P. Mendenhall**

"EXPRESS MAIL" Mailing Label Number: EL656797508US
Date of Deposit: May 22, 2001

HEURISTIC FOR GENERATING OPPOSITE INFORMATION FOR INCLUSION IN FOCUS EVENTS

Background of Invention

Field of the Invention

[0001] The invention relates generally to windowing toolkits for computers.

Background Art

[0002] The basic functionality of a computer is dictated both by the hardware of the computer and by the type of operating system it uses. Various operating systems exist in the marketplace, including Solaris from Sun Microsystems, Inc., MacOS from Apple Computer, Inc., the "Windows" operating systems, *e.g.*, Windows® 95/98 and Windows NT®, from Microsoft Corporation, and Linux. A given combination of computer hardware, an operating system, and a windowing system will be referred to herein as a "platform." Prior to the popularity of the Internet, software developers wrote programs specifically designed to run on specific platforms. Thus, a program written for one platform could not be run on another. However, the advent of the Internet made cross-platform compatibility a necessity.

[0003] Prior art Figure 1 illustrates a conceptual arrangement wherein a first computer **3** running the Solaris platform and a second computer **5** running the Windows® 98 platform are connected to a server **9** via the Internet **7**. A resource provider using the server **9** might be any type of business, governmental, or educational institution. The resource provider has a need to provide its resources to both the user of the Solaris platform and the user of the Windows® 98 platform, but does not have the luxury of being able to custom-design its content for the individual platforms.

[0004] The Java™ programming language was developed by Sun Microsystems to address this problem. The Java™ programming language was designed to be simple for

the programmer to use, yet to be able to run securely over a network and work on a wide range of platforms.

[0005] Prior art Figure 2 illustrates how to create a Java™ application. In order to create a Java™ application, the developer first writes the application in human-readable Java™ source code. As used herein, the term "application" refers to both true Java™ applications and Java™ "applets," which are essentially small applications usually embedded in a web page. In the example shown, the application "Program" **11** is created as a human-readable text file. The name of this text file is given the required extension ".java ".

[0006] A Java™ compiler **13**, such as "javac" available from Sun Microsystems, Inc., is used to compile the source code into a machine-readable binary file **15**. The source text file **11** will contain Java™ language commands, *e.g.*, "import java.awt.Frame". A discussion of the Java™ language itself is beyond the scope of this document. However, complete information regarding the Java™ programming language is available from Sun Microsystems, both in print and via the Internet at java.sun.com. The resulting binary file **15** will automatically receive the same file name as the source text file **11**, but will use ".class" as the trailing extension.

[0007] The Java™ runtime environment incorporates a Java™ "virtual machine" ("JVM") **16** to convert the ".class" byte codes into actual machine executions **17**. The machine executions (like drawing windows, buttons, and user prompt fields) will occur in accordance to the application developer's code instructions. Because Sun Microsystems specifically designed the JVM to run on different platforms, a single set of ".class" byte codes will execute on any platform where a JVM has been installed. An Internet browser such as Netscape Navigator or Microsoft Internet Explorer that incorporates a JVM is called a "Java™-enabled" browser.

[0008] The cross-platform architecture of the Java™ programming language is illustrated in prior art Figure 3, which shows how the Java™ language enables cross-platform applications over the Internet. In the figure, the Solaris platform **3** and the

Windows® 98 platform **5** are each provided with a Java™ virtual machine ("JVM") **21**. The resource provider creates a Java™ application using the Java™ software development kit ("SDK") **23** and makes the compiled Java™ byte codes available on the server **9**. Through standard Internet protocols, both the computer **3** and the computer **5** may obtain a copy of the same byte codes and, despite the difference in platforms, execute the byte codes through their respective JVMs.

[0009] Typical computer applications, including most Java™ applications, provide graphical user interfaces, or GUIs. A GUI consists of graphical components, such as windows, buttons, and text fields displayed on the screen. The user interacts with an application by means of the GUI, clicking on the buttons or typing text into the text fields.

[0010] Platforms, including the Java™ platform, provide the developer convenient means for writing the GUI portions of applications in the form of user interface toolkits. Such toolkits typically include a set of pre-built graphical components (buttons, text fields, etc.) that the developer uses to build applications. The toolkits may also provide mechanisms for other functions. One such function is keeping track of which component will receive keyboard input typed by the user. Typically, at any given time, keyboard input will be directed to one special component, called the "focused component" or "focus owner". This component may be distinguished in appearance by a highlight or a blinking caret. The user may change which component is the focused component, typically by using the mouse to click on the desired new focus owner. Many user interface toolkits will interpret such mouse clicks and respond by resetting the focus owner to the clicked-on component.

[0011] Modern platforms provide facilities for multiple graphical applications to be running at the same time, and each application may present the user with multiple windows. Therefore, a typical display will show many windows simultaneously. One of these windows will usually be distinguished, typically with a darkened titlebar, as the "active window". The active window is the window with which the user is currently interacting. It will contain the focused component, if there is one.

[0012] Prior art Figure 4 illustrates an exemplary display on a screen **31** including windows **33**, **34**, and **35**. Each window includes a title bar **37** for displaying the title of the window and, if applicable, a menu bar **39** containing a number of pull down menu buttons defined by the developer. In this example, window **34** is the active window, as indicated by its darkened title bar. Windows **33** and **35** are inactive as indicated by their grayed out title bars. The text field **61** in window **34** is the focus owner, as indicated by the caret (which may be blinking, to further draw the user's attention). The window **33** includes a number of typical components, including "radio buttons" **41** which in this case allow the user to select a prefix, a text field **43** for entering a name, and an address field **45** for entering an address. Component **47** is a "chooser" that allows the user to choose a state. "Check boxes" **49** allow the user to select one or all of the options that apply. Associated with these check boxes are additional radio buttons **51** and **53** that allow the user to select a desired means of transmission. If the "QUOTE" check box **49** is selected and the telephone radio button is selected, the window **34** appears allowing the user to enter telephone numbers. An additional text area **57** is associated with the "OTHER" check box **49**. Finally, "SUBMIT" and "RESET" buttons **59** are provided to allow the user to either submit the form or to reset it.

[0013] The Java™ platform provides the developer with two user interface toolkits that may be used to build applications: the Abstract Windowing Toolkit, abbreviated AWT, and Swing. The AWT has a unique architecture, in that it is built on top of each platform's native toolkit and uses each platform's native components. For example, an AWT text field consists of the native toolkit's text field component, together with additional data. The underlying native component, called the "heavyweight peer," is used to provide much of the AWT component's functionality. For example, the AWT delegates the job of painting the component on the screen to the native toolkit. In this way, the AWT can be used to build applications that, on each platform, look and behave like the platform's native applications.

[0014] Swing, by contrast, contains no heavyweight peers. Instead, its components are "lightweight," that is, have no corresponding native components. In fact, the underlying

native toolkit is unaware of Swing's components, so nearly all of the components' functionality must be provided by Swing.

[0015] When a user interacts with a computer by typing on the keyboard or clicking the mouse on different areas of the computer screen, the underlying native platform informs the appropriate application of the user's actions by means of native "events." These events are platform-specific and contain different information depending on the action that the user performed. For example, if the user typed a key on the keyboard, the underlying platform might generate a "key pressed" event when the key was pressed and a "key released event" when the key was released. The events will contain various information about the user action, such as which key was pressed and released or the state of the keyboard (*e.g.*, the CAPS-LOCK key) during the user's actions.

[0016] As mentioned above, the events are generated by the underlying platform and are therefore platform-specific. Different platforms will generate different events in response to the same user actions, and the events themselves will contain different information depending on the platform that generated them. Another difference between platforms may be the way in which events are delivered to the appropriate application. On some systems, events might be placed on a queue, and it is the application's responsibility to dequeue the events and process them. On other systems, the application may register a special procedure, called an "event handler," with the underlying platform. This event handler will be called whenever the platform wishes to deliver an event to that application.

[0017] These platform differences in events and event delivery mechanisms are some of the reasons that, prior to the Java™ platform's introduction, it was impossible for developers to write applications that worked on multiple platforms without customizing the application for each platform. The Java™ user interface toolkits address this problem by providing a uniform event model for all platforms on which the Java™ platform is implemented. The Java™ implementation hides both the native delivery mechanism and the native events themselves from its applications by registering native handlers or dequeuing native events as appropriate. Then, based on the native events it receives, it

generates the appropriate "Java™ events" and delivers them to its applications via a mechanism of its own (typically by calling Java™ event handlers registered by the Java™ application.)

[0018] Because different platforms generate different native events, it follows that there is not a one-to-one mapping between native events and Java™ events. Also, because native events on different platforms contain different information, in some cases platform-specific information may be omitted from a Java™ event, while in other cases information not present in a native event may need to be computed for inclusion in a Java™ event. It is the job of the Java™ implementation on each platform to unify these differences so that Java™ applications on different platforms receive the same sequence of Java™ events when exposed to the same user actions.

[0019] One class of Java™ events generated by the Java™ implementation on each platform are focus events. A component becomes the focus owner when it receives a FocusGained event, and it ceases being the focus owner when it receives a FocusLost event. The Java™ Standard Edition SDK, version 1.4 defines a new field in its focus events: the "opposite" field. In a FocusLost event, the opposite field specifies the component that is gaining focus in conjunction with this FocusLost event, that is, it specifies where the focus is going next. In a FocusGained event, the opposite field specifies the component that is losing focus in conjunction with this FocusGained event, that is, it specifies where the focus is coming from. Some native platforms, such as those running the various Windows operating systems, provide the opposite components in their native focus events, and those components can then be included in the corresponding Java™ events. However, the X windowing system, for example, does not provide this information, so Java™ implementations on X-based platforms must compute the opposite components for inclusion in the Java™ focus events.

[0020] Therefore, there is a need for a method for computing the information to include in opposite fields of Java™ focus events.

Summary of Invention

[0021] In one aspect, the invention relates to a method for generating information for inclusion in focus events which comprises maintaining a list of components requesting focus in a selected application and determining whether a target of a first focus event matches a component at the head of the list. If the target of the first focus event matches the component at the head of the list, the method further comprises marking the component at the head of the list for inclusion in an opposite field of a second focus event.

[0022] In another aspect, the invention relates to a method for generating information for inclusion in focus events which comprises maintaining a list of components requesting focus in a selected application and determining whether a target of a first focus event matches a component at the head of the list. If the target of the first focus event matches the component at the head of the list, the method further comprises marking the component at the head of the list for inclusion in an opposite field of a second focus event and marking a component next to the component at the head of the list for inclusion in an opposite field of the first focus event.

[0023] In another aspect, the invention relates to a computer-readable medium having stored thereon a program which is executable by a processor. The program comprises instructions for maintaining a list of components requesting focus in a selected application. The program further includes determining an opposite field of a first focus event and an opposite field of a second focus event based on a target of the first focus event, a target of the second focus event, and the list of components requesting focus.

[0024] Other aspects and advantages of the invention will be apparent from the following description and the appended claims.

Brief Description of Drawings

[0025] Figure 1 illustrates a multiple platform environment.

[0026] Figure 2 illustrates a mechanism for creating Java™ applications.

[0027] Figure 3 illustrates a Java™ application running in a multiple platform environment.

[0028] Figure 4 illustrates a typical graphical user interface (GUI).

[0029] Figure 5 illustrates a typical computer and its components as they relate to the Java™ virtual machine.

[0030] Figure 6 is a graphical representation of a Focus List according to one embodiment of the invention.

[0031] Figure 7 is a flowchart illustrating how list elements are added to the Focus List shown in Figure 6.

[0032] Figure 8A is a flowchart illustrating how the opposite field for a FocusLost event is determined in accordance with one embodiment of the invention.

[0033] Figure 8B is a continuation of Figure 8A.

[0034] Figure 9 is a flowchart illustrating how the opposite field for a FocusGained event is determined in accordance with one embodiment of the invention.

Detailed Description

[0035] Specific embodiments of the invention will now be described in detail with reference to the accompanying drawings. Like elements in the various figures are denoted by the same reference numerals for consistency.

[0036] The invention described here may be implemented on virtually any type of computer regardless of the platform being used. For example, as shown in Figure 5, a typical computer 71 will have a processor 73, associated memory 75, and numerous other elements and functionalities typical to today's computers (not shown). The computer 71 will have associated therewith input means such as a keyboard 77 and a mouse 79, although in an accessible environment these input means may take other forms. The computer 71 will also be associated with an output device such as a display 81, which may also take a different form in an accessible environment. Computer 71 is connected

via a connection means 83 to the Internet 7. The computer 71 is configured to run a JavaTM virtual machine 21, implemented either in hardware or in software.

[0037] The present invention provides a method for computing the information to include in "opposite" fields of Java™ focus events. The method works perfectly for computing such information whenever focus is transferred between components within the same top-level window. When focus transfers outside of the window, the method may fail and report the opposite component incorrectly or as "null". However, it will recover and report opposite components correctly upon subsequent, intra-window transfers.

[0038] The method relies on two observations about the circumstances under which JavaTM focus events are generated due only to the operation of the JavaTM application in question. The key observation is that such events are generated only as a result of one of two causes: either a JavaTM-level programmatic focus request, or a user button click on a focusable heavyweight component (resulting in a native focus request on that component). In each of these two cases, a pair of events is generated: a FocusLost event on the component that previously had focus, and a FocusGained event on the component requesting focus. Thus, our second observation is that, since application-caused JavaTM focus events are always generated in such "lost/gained" pairs, computing the opposite component for FocusGained events is easy: it is the component on which a FocusLost event has just been generated. If there is no such FocusLost event, then focus is coming from somewhere outside the scope of our application; in that case, we use "null" as the opposite component.

[0039] On the other hand, in order to compute the opposite component for a FocusLost event, we would need to predict the future: we would need to know what FocusGained event will be generated next. We can't know this information for certain – for example, the focus change may not be internal to the application and focus may be going to an unrelated, native application window. Recall, however, that each focus request will typically result in a FocusGained event being generated. Thus, if we keep a queue of all the focus requests, we can use it to guess the opposite component for FocusLost events. When generating a FocusLost event, we would look at the first request on the queue, use

the component making the request as the opposite component in the FocusLost event, and dequeue the request.

[0040] In order to compute this information, a list of components that have issued either Java™ or native-level focus requests, but have not yet received focus notification events, is maintained. Herein, this list of components is referred to as the Focus List. Figure 6 shows a graphical representation of the Focus List, generally identified by reference numeral 100. Focus List 100 can have zero, one, or more list elements 102. Each list element has a "requestor" member and a "next" member. The "requestor" member contains data that identifies a Java™ component 105 that has at some point in time issued either a Java™ or native-level focus request. The "next" member contains the memory location of the next element in the list. Two pointers called "Focus List Head" and "Focus List End" are maintained. Focus List Head points to the top of Focus List 100, and Focus List End points to the end of Focus List 100.

[0041] Figure 7 is a flowchart that illustrates the process for adding list elements (102 in Figure 6) to the Focus List (100 shown in Figure 6). A new element is added to the Focus List whenever either a native-level focus request or a Java™ focus request is issued. In the native request scenario, a user clicks on a heavyweight focusable component (ST106), which results in the component receiving a native-level "button pressed" event (ST110) and in the underlying platform issuing a native-level focus request on behalf of the component. In the Java™ request scenario, a Java™ component issues a programmatic focus request (ST108) through a function invocation.

[0042] As illustrated, the process involves checking whether Focus List End is null (ST112), i.e., whether Focus List (100 in Figure 6) is empty. If Focus List End is null, then memory allocation is made for a new list element (ST114). At step ST116, the new list element is added to the Focus List (100 in Figure 6). Then, Focus List End is modified such that it points to the new list element. At step ST118, the "requestor" member of the element pointed to by Focus List End is set to the component requesting focus, and the "next" member of the element pointed to by Focus List End is set to null.

[0043] Returning to step ST112, if Focus List End is not null, then the process involves checking whether the component requesting focus is the same as the "requestor" member of the element pointed to by Focus List End (ST120). If the component requesting focus and the "requestor" member of the element pointed to by Focus List End are the same, then no action is required (ST122). Otherwise, memory allocation is made for a new list element (ST124). The "next" member of the element pointed to by Focus List End is set to the new list element, and Focus List End is then adjusted to point to the new list element (ST126). The "requestor" member of the element pointed to by Focus List End is set to the component requesting focus, and the "next" member of the element pointed to by Focus List End is set to null (ST118).

[0044] As Java™-level focus events are generated by the Java™ platform, the opposite component involved in the focus transfer is computed. Figure 8A shows how to compute the opposite component when a FocusLost event is being generated for the component that currently has the focus (ST128). At this point, the process of determining the opposite component involves checking whether Focus List Head is null (ST130). If Focus List Head is null, there are no elements in the Focus List (100 in Figure 6), and the opposite component for the FocusLost event is set to null (ST131), because no guess can be made as to where the focus is going (it is probably going out of the scope of this application). If Focus List Head is not null, the process involves determining whether the current focus owner matches the component at the head of the Focus List (100 in Figure 6). If it does not, or if there are no components in the Focus List, then the FocusLost event also resulted from a focus request from outside of the current application, such as a user clicking on an unrelated window on the desktop. In this case, the opposite component for the FocusLost event is set to null. Then the Focus List (100 in Figure 6) is cleared, because, once focus leaves the application, the queued up requests will be ignored and will not be resulting in focus events.

[0045] To clear the Focus List (100 in Figure 6), the "next" member of the element pointed to by Focus List Head is copied into a temporary variable (ST132). The memory allocated to the list element pointed to by Focus List Head is then de-allocated (ST134). After this, Focus List Head is modified to point to the list element identified in the

temporary variable (ST136). The process then checks whether Focus List Head is null (ST138). If Focus List Head is not null, steps ST132, ST134, and ST136 are repeated until Focus List Head becomes null. When Focus List Head becomes null, the opposite component for the FocusLost event is set to null (ST140).

[0046] Returning to step ST133, if the "requestor" member of the list element pointed to by Focus List Head is the same as the current focus owner, then the component identified by the "requestor" member is saved as the opposite field for the next FocusGained event. Figure 8B illustrates the process in detail. As shown, the "next" member of the list element at the head of the Focus List (100 in Figure 6) is copied into a temporary variable, and the "requestor" member of the list element is copied into a variable called "forGained" (ST142). Then the memory allocated to the element at the head of the Focus List (100 in Figure 6) is de-allocated (ST144). Focus List Head is then modified to point to the list element identified in the temporary variable (ST146). The process continues with checking whether Focus List Head is null (ST148). If Focus List Head is null, then Focus List End is set to null (ST150), and the opposite component for the FocusLost event is set to null (ST152). If Focus List Head is not null, then the opposite component for the FocusLost event is set to the "requestor" member of the list element pointed to by Focus List Head (ST154).

[0047] Figure 9 illustrates how the opposite component for FocusGained events is generated (ST156). Focus List Head is first examined to see if it is null (ST158). If Focus List Head is null, this indicates that the FocusGained event is the result of something external to this application, and the opposite component for the FocusGained event is set to null (ST159). If Focus List Head is not null, the process involves checking whether the new focus owner matches the component at the head of the Focus List (ST160). If the new focus owner matches the component at the head of the Focus List (100 in Figure 6), the opposite component for the FocusGained event is set to the component identified in the forGained variable (ST162).

[0048] Returning to step ST160, if the component at the head of the Focus List (100 in Figure 6) does not match the new focus owner, then the FocusGained event is being

generated on a component for which we are not expecting such an event. This may happen if, for example, focus had been transferred out of the scope of this application before all the focus events for the queued up requests had been generated, and is now being transferred back. This case requires the Focus List (100 in Figure 6) to be cleared, because focus events corresponding to the requests on the list will not be generated. To clear the list, the "next" member of the list element at the head of the Focus List (100 in Figure 6) is copied into a "temporary" variable (ST164). Then, the memory allocated to this list element is de-allocated (ST166). Focus List Head is modified to point to the list element identified by the temporary variable (ST168). At step ST170, the process further involves checking whether Focus List Head is null. If Focus List Head is not null, steps ST164, ST166, and ST168 are repeated until Focus List Head becomes null. When Focus List Head becomes null (ST172), Focus List End is set to null (ST174), and the opposite component for the FocusGained event is set to null (ST176).

[0049] The invention may provide general advantages in that it provides a method for computing the information required for opposite fields of focus events. The invention is useful when the native platform or native windowing toolkit does not normally provide this information. As described above, a list of components that have issued focus requests is maintained. The list is then used to determine the opposite information when focus events are processed.

[0050] While the invention has been described with respect to a limited number of embodiments, those skilled in the art, having benefit of this disclosure, will appreciate that other embodiments can be devised which do not depart from the scope of the invention as disclosed herein. Accordingly, the scope of the invention should be limited only by the attached claims.